COMPUTING GRAPH METRICS AND GRAPH PROPERTIES WITH SQL QUERIES

A Thesis Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Xiantian Zhou

August 2019

COMPUTING GRAPH METRICS AND GRAPH PROPERTIES WITH SQL QUERIES

Xiantian Zhou

APPROVED:

Dr.Carlos Ordonez
Dept. of Computer Science

Dr.Gopal Pandurangan
Dept. of Computer Science

Dr.Robert Azencott
Dept. of Mathematics

Dr. Dan E. Wells, Dean, College of Natural Sciences and Mathematics

Acknowledgement

First of all, I thank my advisor, Dr.Carlos Ordonez, for his valuable guidance and patience through my academic life in University of Houston. I would like to thank the committee members, Dr.Gopal Pandurangan, Dr.Robert Azencott, for sharing their knowledge and experience. Also, I appreciate the feedback and technical support from my research group partner, Sikder Tahsin Al-Amin. Thanks to my husband, Hu Li, for his loving care and support.

COMPUTING GRAPH METRICS AND GRAPH PROPERTIES WITH SQL QUERIES

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Xiantian Zhou

August 2019

Abstract

Within big data analytics, graph problems are as important as machine learning. There exist many algorithms to analyze large graphs, but they are limited by main memory. On the other hand, a lot of data stored on DBMSs needs to be analyzed as graphs. Moreover, DBMSs can work in parallel, and they do not have RAM limitations. In this paper, we propose several algorithms that produce metrics and show properties of the graph as well as help us to understand the graph structure specifically diameter and betweenness centrality. This work is a big step beyond transitive closure and recursive queries. We propose optimized SQL queries that work on a graph stored in relational form as triples which can compute diameter and betweenness centrality in a more flexible and efficient manner. We study how to optimize SQL queries combining demanding joins and aggregations that remove main memory limitation and also work in parallel. Finally, we provide an experimental evaluation to understand accuracy and performance. We compare our algorithms with popular platforms including Python and Spark. We experimentally show our that SQL algorithms are accurate and efficient.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Graphs are ideal for representing relationships and are increasingly used as primary data structures for representing interconnected data. Large-scale graphs have been applied in many emerging areas. Typical graph can be social networks, roads connecting cities, telecommunication, flights linking airports. Since graphs are important, we need to analyze them to understand their structure. But understanding the structure of the graph is more difficult and complex than solving fundamental graph problems like exploring the graph [2] and transitive closure [11]. The problem is hard and very important in graph theory. Graph analytics remains one of the most computationally intensive tasks in big data analytics due to large graph size, the complex structure of the graph, and patterns presented in the graph. Especially, when the graph grows so fast they do not fit in main memory.

There are many different graph metrics we could use to analyze the structure of graphs, such as vertex degree or reachability. These are statistics of vertex or edges and can be obtained by doing transitive closure. There are other complicated metrics which are statistics of paths such as betweenness centrality and diameter. Betweenness centrality is calculated as the fraction of shortest paths between vertex pairs that pass through the vertex of interest. It is a measure of the inuence a vertex has over the spread of information through the network. Diameter is the longest shortest path of the graph. It is an index measuring the extent of the graph. Those two metrics are important to understand the graph and to find which vertex is more important when information is delivered. These metrics help solve other harder graph problems such as triangles detection. Here, we choose diameter and betweenness centrality as our topic.

Relational databases remain the most common technology to store transactional and analytical databases, due to optimized I/O, robustness, and security control. A lot of data stored on DBMSs (database management system) can be potentially analyzed as graphs [19]. Even though the data is not in a DBMS, it is fast to load a large data set into a DBMS. Graphs can be represented in terms of database perspective. However, processing large graphs in a large scale distributed system has not received much attention in DBMS using relational queries.

With existing DBMSs, we revisit the problem of solving graph algorithms with SQL queries. SQL queries are concise, efficient, and they are used to solve many problems. We propose several algorithms solved with SQL queries that can help

understand the structure of a graph. We perform our experiments on columnar parallel DBMS with *shared-nothing* architecture. While our queries can work in any DBMS, it is experimentally shown that columnar and array DBMSs performance is substantially better than row DBMSs for graphs analysis [22]. Also, parallel database systems have a significant performance advantage over Hadoop MapReduce in executing a variety of data-intensive analysis benchmarks [23]. Our goal is to prove that DBMS can help us to understand graph structure with something more complicated than what was done before with recursive queries (transitive closure) [13]. Moreover, columnar DBMS provides significant accelerations in JOIN and Group By queries.

1.2 Our Contributions

In our work, we propose several algorithms that produce metrics and show properties such as graph diameter and the betweenness centrality of a vertex . We study how to express the computation of these algorithms only with relational queries and how we can optimize the queries. Most of the existing graph databases fail when the data volume is too large. Also, the usability of these graph databases is comparatively less than relational DBMSs since they have no standard set of rules. Moreover, these graph databases along with many popular platforms including Python and Spark are limited by main memory. We believe efficient graph algorithms for relational databases will avoid wasting time exporting data or setting up external systems. In our opinion, even though query optimization is classical and well-studied, topic optimization of relational queries on graphs needs further research.

1.3 Thesis Organization

The organization of this thesis is presented below:

Chapter 2 is a literature review of graph-analytics computation. In Chapter 3 we present notation, denitions, and background information about graph analytics. We program our algorithms with SQL queries and optimize those queries. Chapter 4 explains details about our algorithms and optimizations. We also analyze the time complexity of our algorithms in Chapter 4. In Chapter 5, we present experimental proof of our algorithms, the impact of the optimizations, and compare the performance of our algorithms with other popular graph analysis platforms.

Chapter 2

Related Work

We introduce the application of graphs, the importance of graph analytics, the reason that we pick up this topic and our contribution in the first chapter. In this chapter, we will explain the related work of graph analytics, betweenness centrality and diameter, and database systems.

2.1 Graph Analytics

A lot of research has been done in the field of graph analytics. Recent work on graphs offers a vertex-centric query interface to express many graph queries [17]. A novel method called core labeling was proposed to handle reachability queries for massive, sparse graphs [13]. Abughofa et al., 2018 studied processing dynamic graphs in real-time and proposed an end-to-end framework which allowed graph updates in real-time and supported efficient complex analytics in addition to online

transaction processing (OLTP) queries. However, querying from large graphs stored on a DBMS using relational queries has not received much attention. Malewicz et al., 2010 proposed a system named Pregel for large-scale graph processing. Pregel is designed for sparse graphs where communication occurs mainly over edges, and the entire computation state resides in RAM. How relational database management systems (RDBMSs) can support graph processing at the SQL level was revisited in [26]. The authors proposed new relational algebra operations.

SQL recursive queries are a fundamental mechanism to analyze graphs in a DBMS, whose processing and optimization is significantly harder than traditional SPJ (select, projection, and join) queries. Al-Amin et al., 2018 showed SQL queries on a graph stored in relational form as triples could reveal many interesting properties and patterns on the graph in a more flexible manner and more efficient than existing systems. The linearly recursive queries, can summarize interesting patterns including reachability, paths, and connected components. Exploratory queries can be efficiently evaluated based on the input edges. Other languages other than SQL queries for processing graphs were also studied. Gremlin is a graph traversal language and machine that provides a common platform for supporting any graph computing system [24]. Thakkar et al., 2017 presented a formalization of graph pattern matching for Gremlin language.

2.2 Diameter and Betweenness Centrality

2.2.1 Diameter

The diameter is the longest shortest path (the longest graph geodesic) of a graph. It is a fundamental graph parameter, and its computation is necessary for many applications. The fastest known way to compute the diameter exactly is to solve the All-Pairs Shortest Paths (APSP) problem. And many attempts were made to seek efficient algorithms that approximate the diameter. Two solutions were presented which could achieve a better approximation for the diameter, one running in $O(m^{3/2})$ time and the other running in $O(mn^{2/3})$ where m is the number of edges and n is the number of vertices in [12]. The asymptotic growth of the diameter of a graph obtained by adding sparse long edges was studied in [9].

2.2.2 Betweenness Centrality

Betweenness centrality is a centrality measure based on shortest paths, widely used in complex network analysis. It is essential in the analysis of graphs, but costly to compute. Since it was introduced independently by Anthoisse and Freeman in [4], [15], many works have been done about making it faster. Brandes, 2001 developed a fast algorithm that runs in $O(n+m)$ on unweighted graph and $O(mn+n^2log(n))$ time on weighted graphs, where n is the number of vertices and m is the number of edges in the graph [10]. These are also the worst case time bounds for computing the exact value of the betweenness centrality score. Recently, many works focusing on how to obtain

rough approximations of betweenness centrality have been done. Bader et al., 2007 presented a novel approximation for computing betweenness centrality of a given vertex, for both weighted and unweighted graphs. The approximation algorithm was based on an adaptive sampling technique that significantly reduced the number of single-source shortest path computations for vertices with high centrality. The random sampling algorithm gave good betweenness approximations for biological networks, road networks, and web crawls.

There are some work has been done on analyzing graphs on distributed platforms. Bertolucci et al., 2010 proposed a solution that estimates the current flow betweenness in a distributed setting. The solution was based on the Gather Apply Scatter model that estimates the current flow betweenness in a distributed setting using the Apache Spark framework. The experimental evaluation showed that the algorithm achieved a high correlation with the exact value of the index and outperformed other algorithms. Other works such as getting betweenness centrality for complex graphs has been done in [7]. Geisberger et al., 2008 proposed a framework for an unbiased approximation of betweenness centrality that generalized a previous approach by Brandes, 2010. The new schemes yielded significantly better approximation than before for many real-world inputs and obtained good approximations for the betweenness centrality of unimportant nodes.

2.3 Database Systems

Parallel DBMS and Spark are popular database platforms for graph analytics. Some works have been tried on analyzing graph using database platforms.

2.3.1 Parallel DBMS

There are different kinds of DBMSs. Columnar DBMSs are a faster class of database systems, with significantly different storage and query processing mechanisms compared to row DBMSs which is still the dominating technology. Ordonez et al., 2017 studied the optimization of recursive queries on a columnar DBMS focusing on two fundamental and complementary graph problems: transitive closure and adjacency matrix multiplication, and presented comprehensive experiments comparing recursive query processing on columnar, row, and array DBMSs to analyze large graphs with different shape and density. And it showed a columnar DBMS with tuned query optimization was uniformly faster than row and array systems to analyze large graphs, regardless of their shape, density, and connectivity. On the other hand, there was no clear winner between the row and array DBMSs. So we choose a columnar DBMS as the platform to run our queries for my thesis. Columnar DBMS is a parallel processing platform, and provides significant accelerations in JOIN and Group By queries.

2.3.2 Spark

Spark is a unified analytics engine for large-scale data processing, and is a scalable data processing platform. There has been much research work done about how to analyze data using Spark. Cho et al., 2019 proposed a new model to detect communities in a graph using Spark. Alemi et al., 2017 presented a distributed MapReduce-based algorithm being executed on Apache Spark, called CCFinder, to efficiently compute clustering coefficient in very big graphs. And also they proved that the algorithm could efficiently detect existing triangles through using the proposed data structure which was cached in the distributed memory provided by Spark and reused multiple times. Balaji et al. 2016 demonstrated the use of Spark for iterative graph path queries. The RDD abstraction of Spark made it possible by providing a persistent storage platform for repeated processing of data. They also presented several variations for path query processing using Spark. Naacke et al., 2017 studied the use of two distribute join algorithms, partitioned join and broadcast join, for the evaluation of basic graph pattern expressions on top of Apache Spark. The results showed that hybrid join plans introduced more flexibility and often achieved better performance than single kind join plans.

Chapter 3

Background

This section explains works related to graph analytics, betweenness centrality, diameter, and database systems that were discussed in the previous chapter. Numerous work has been done concerning graphs, including proposing new algorithms, and lowering the time complexity. Some work concerning analyzing large-scale graphs using distributed platforms has been tried. But most computation of those algorithms resides in RAM. In this chapter, the background of our work is introduced.

3.1 Problem Definitions

3.1.1 Graph

Let $G = (V, E)$ be a directed graph with $n = |V|$ vertices and $m = |E|$ edges, where V is a set of vertices and E is a set of edges, considered as an ordered pairs of vertices.

An edge in E links two vertices in V and has a direction. This definition allows the presence of cycles and cliques in graphs. A cycle is a path which starts and ends at the same vertex. A clique is a complete sub-graph of G. The adjacency matrix of G is a $n \times n$ matrix such that the cell i, j holds 1 when there exists an edge from vertex i to vertex j.

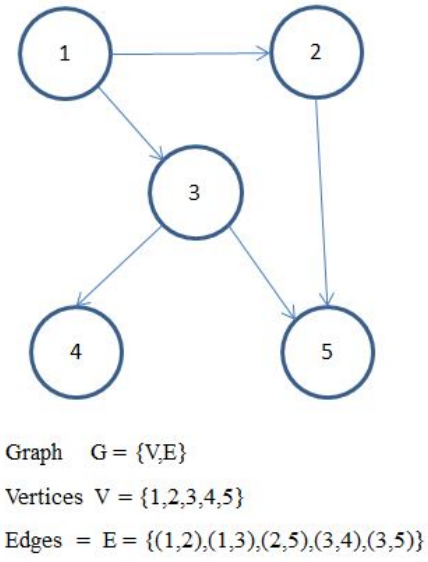

Figure 3.1: A graph example, the graph G.

From a database perspective, graph G is stored in table E as a list of edges (adjacency list). Let, table E be defined as $E(i, j, v)$ with primary key (i, j) representing the source and destination vertices and v representing a numeric value, e.g., weight. A row from table E represents the existence of an edge. So, the graph should be a sparse graph, with only existing edges present in the table. For an undirected graph, for each row (i, j, v), this study adds another row (j, i, v) so that all the edges are

bidirectional. In summary, from a mathematical point of view, E is a sparse matrix. From a database perspective, E is a large and narrow table having one edge per row. For example, if city names are depicted in graphs, then i and j will be city names and v can be the distance or travel cost between them. However, if phone calls are graphed, then E will have multiple edges per person pair, not one edge per phone call or message. According to this, the data set for graph G is as depicted in Table 3.1.

Table 3.1: The dataset for graph G.

i	j	v
1	2	1
1	3	1
2	5	1
3	4	1
3	5	1

3.1.2 Diameter

The diameter of a graph is the length of the longest shortest path between any two graph vertices (i, j),

$$d = max_{(i,j)}d(i, j)$$

where $d(i, j)$ is a graph distance. $d(i, j)$ is defined as the number of edges in a shortest path connecting i and j. The diameter of a graph G is d that is the maximum distance between any pair of vertices in the graph. As Figure 3.2 shows, the diameter is an

index measuring the extent of the graph.

(a)

(b)

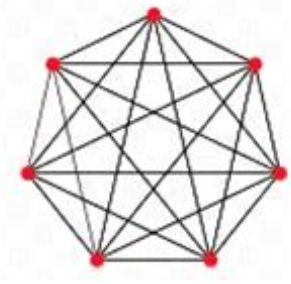

Figure 3.2: Graphs with different diameters.

3.1.3 Betweenness Centrality

Betweenness centrality is an important class of centrality that measures the extent to which a vertex lies on the paths between others. Top-K vertices with the high betweenness centrality values are very important in analyzing graphs. As Figure 3.3 demonstrates, users with high betweenness centrality in online social networks are instrumental for the spread of information because these nodes are present on many of the shortest paths in a graph. A path is defined from $i \in V$ to $j \in V$ as an alternating sequence of vertices and edges, starting from i and ending with j, such that each edge connects its previous and next vertex. Let $\varphi_{ij} = \varphi_{ji}$ denote the number of shortest paths from i to j, where $\varphi_{ii} = 1$ by convention. And $\varphi_{ij}(m)$ is the number of shortest paths from i to j that $m \in V$ lies on. The betweenness centrality for vertex m is $C_B(m) = \sum_{i \neq m \neq j \in V} \frac{\varphi_{ij}(m)}{\varphi_{ij}}$. Define the pair-dependency $\delta_{ij}(m)$ as

$\frac{\varphi_{ij}(m)}{\varphi_{ij}}$, then the standard formula for betweenness centrality can be written as:

$$C_B(m) = \sum_{i \neq m \neq j \in V} \delta_{ij}(m). \tag{3.1}$$

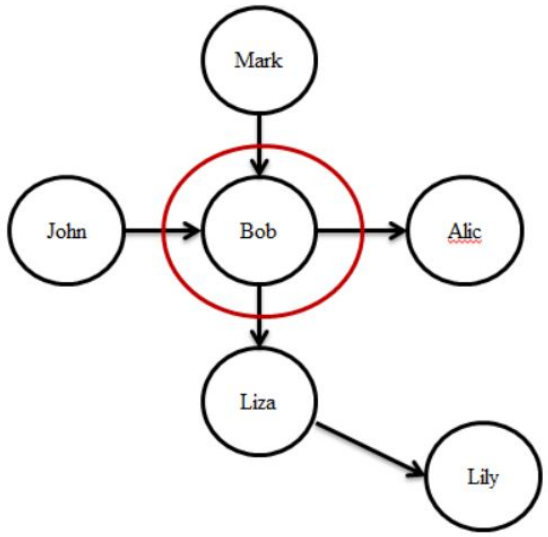

Figure 3.3: One node with high betweenness centrality.

3.2 Parallel Processing

Vertica is used as the platform to run our solution. It is a column-oriented analytic platform designed to manage large, fast-growing volumes of data and provide very fast query performance when used for query-intensive applications. This columnar DBMS with shared nothing architecture offers parallel processing. First Data is partitioned into independent nodes. Partitioning helps organize data on each node into

different storage containers, reduces I/O and improves query performance. Vertica uses massively parallel processing (MPP) architecture to distribute queries on independent nodes and scale performance linearly. So each node will run queries on its data.

Chapter 4

Programming Algorithms with Queries

The previous chapter discussed the definition of graph datasets, betweenness centrality, diameter, and parallel DBMS.

This study's main technical contributions are presented in this chapter. Several algorithms are developed (diameter and betweenness centrality) using SQL that produce metrics and show properties of the graph as well as help us to understand the graph structure. Diameter is the longest shortest path for connected graphs. To get the correct betweenness centrality value for every vertex, it is necessary to know the longest shortest path to specify the depth of JOIN.

4.1 General Algorithm

4.1.1 Diameter

The exact value of the diameter of a graph is achieved by calculating all pairs' shortest paths (APSP). The algorithms for the APSP problem include matrix multiplication or repeated squaring, the Floyd-Warshall algorithm, and transitive closure of a graph. Figure 4.1 shows the transitive closure algorithm.

Algorithm 1: Transitive Closure

```
n ← |V|;
T_ij(0) ← 0, i, j ∈ V;
for i ← 1 to n do
    for j ← 1 to n do
        if i = j or (i, j) ∈ E then
            T_ij(0) ← 1
        end
    end
end
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            T_ij(k) ← T_ij(k − 1) ∨ (T_ik(k − 1) ∧ T_kj(k − 1))
        end
    end
end
return T(n) ;
```

Figure 4.1: Transitive Closure algorithm.

4.1.2 Betweenness Centrality

The best known and fastest algorithm for betweenness centrality was proposed by Brands, 2001. Figure 4.2, which shows the details of this algorithm, demonstrates that the algorithm could be divided into two steps, computing the number of the shortest path between all pair of vertices(including the intermediate vertices) and summing up all pair-dependency. In the first step, a queue Q is used to perform graph traversal, and all the intermediate vertices of different paths are added to different lists. In the second step, all the pair-dependency are summed up in the order of non-increasing distance from the source vertex s.

4.2 Programming with Queries and Optimizations

4.2.1 Programming with Queries

SQL queries remove RAM limitations automatically eliminating any worry about whether datasets can fit in the main memory or not. The detail of this solution is shown below.

4.2.1.1 Diameter

The diameter of a graph is the length of the longest shortest path between any two vertices (i, j). To find the diameter, linear recursive queries could be used as transitive closures did to get all the shortest path of any two vertices first. But

Algorithm 2: Betweenness centrality in unweighted graphs

```
C_B[v] ← 0, v ∈ V;
for s ∈ V do
    S ← empty stack;
    P[w] ← empty list, w ∈ V;
    σ[t] ← 0, t ∈ V;   σ[s] ← 1;
    d[t] ← −1, t ∈ V;   d[s] ← 0;
    Q ← empty queue;
    enqueue s → Q;
    while Q not empty do
        dequeue v ← Q;
        push v → S;
        foreach neighbor w of v do
            // w found for the first time?
            if d[w] < 0 then
                enqueue w → Q;
                d[w] ← d[v] + 1;
            end
            // shortest path to w via v?
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v];
                append v → P[w];
            end
        end
    end
    δ[v] ← 0, v ∈ V;
    // S returns vertices in order of non-increasing distance from s
    while S not empty do
        pop w ← S;
        for v ∈ P[w] do δ[v] ← δ[v] + (σ[v]/σ[w]) · (1 + δ[w]);
        if w ≠ s then C_B[w] ← C_B[w] + δ[w];
    end
end
```

Figure 4.2: The best known and fast algorithm for Betweenness Centrality.

diameter is different since it focuses on the "longest" one of all the shortest path and does not have to get the reachability for every vertex, which is an optimization studied in this section. Linear recursive are queries of the form: $R = R \cup (R \bowtie E)$, where the result of $R \bowtie E$ gets added to R itself. For the diameter, only the longest shortest path matters. Therefore, compute $R_k = R_{k-1} \bowtie E$, and then SELECT the length of the longest shortest path from R_k. Hence, the iterations are of the form $k = 1, 2, 3....$ The base step produces $R_1 = E$. And then $R_2 = E \bowtie E = R_1 \bowtie_{R_1.j=E.i} E$,... and so on. When R_k becomes empty since no more rows satisfy the JOIN condition, the maximum iteration is reached. To avoid cycles in the path, check it using WHERE $R.i! = E.j$. Finding the diameter is difficult as the table size grows dramatically after some depths of JOIN and it will take a lot of time. The join query is shown below.

```
SELECT d+1, R.i, E.j, R.v+E.v
 FROM  R
 JOIN T ON R.j=E.i
 GROUP BY R.i, E.j
 HAVING R.i!=E.j
```

The explanation for why this query works is not difficult. It is possible to get all paths by doing transitive closure and get all the paths whose length is K by doing join k times [21]. So, table R^k contains all the paths whose length are k. When R^k is empty, no more rows satisfy the JOIN condition, the maximum iteration is reached, and the diameter is $k - 1$.

4.2.1.2 Betweenness Centrality

To get the value of betweenness centrality for each vertex, there are two steps: computing the length and number of shortest paths between all pairs of vertices and adding all pair-dependencies.

For the first step, we could do linear recursive queries as illustrated in the diameter part, it seems similar to transitive closure. But betweenness centrality is far more difficult than transitive closure since it needs not only the reachability of each vertex but also the number of shortest paths, and all the intermediate vertices that lie on those paths. One intuitive way to do it is to store the starting vertex, ending vertex and all the intermediate vertices of the path in a row. Then each row represents a path, and the number of columns is determined by the length of the path. As a result, all the temporary tables obtained at different depths of join have different numbers of columns. After doing all the joins, it is necessary to union all the temporary tables together to select the shortest paths for all pairs. The number of columns for the union table is set by the longest path. But space and time efficiency would be very low since most of the paths are much shorter than the longest shortest path. Also, it is difficult to obtain pair-dependency and calculate the betweenness centrality.

We developed one algorithm that first stores all the intermediate vertices in columns, and then changes all the columns into rows before performing UNION ALL. We also created a new column called $'id'$ by using SEQUENCE that was added to each row to uniquely identify each path. So each row includes the path id, the

starting vertex i, the ending vertex j and one intermediate vertex m. After completing the union table $U(id, i, j, m)$ which has all the shortest paths obtained in each depths, we performed the following actions:

$$F1(i, j, s\_path) = \pi_{u,v,count(id)}(U),$$

to get φ_{ij} for the pair (i, j), we then used

$$F2(i, j, m, s\_path\_m) = \pi_{i,j,m,count(id)}(U),$$

to get $\varphi_{ij}(m)$. Finally, we obtained the betweenness centrality for each vertex m using

$$\pi_{F2.m,sum(s\_path\_m/s\_path)}(F1 \bowtie_{F1.i=F2.i \text{ and } F1.j=F2.j} F2),$$

was used to sum up all the pair-dependencies.

```
/* doing JOIN */
CREATE TABLE Rn
 AS SELECT nextval('sequ') as id, t1.i as i, t2.j as j,
           t1.v + t2.v as v,
           t1.m1 as m1,t1.m2 as m2...t1.m(n-1) as m(n-2),  t2.i as m(n-1)
 FROM Rn-1 t1
 JOIN E t2 ON t1.j=t2.i;
```

```
/*changing all the intermediate vertices columns to rows*/
SELECT id as id, i as i, j as j, v as v , m1 AS m
FROM R2
UNION ALL
SELECT id as id, i as i, j as j, v as v , m2 AS m
FROM R3
...
UNION ALL
SELECT id as id, d as d, i as i, j as j, v as v , m(n-1) AS m
FROM Rn;

/*getting the number of the shortest paths for each pair of (i,j)*/
CREATE  TABLE F1
 AS SELECT i AS i, j AS j, count(distinct id)
 FROM U
 GROUP BY i, j;

/*getting the number of shortest paths that pass through vertex m*/
CREATE TABLE F2
 AS SELECT i AS i, j AS j, m AS m, count(distinct id)
 FROM U
 GROUP BY i, j, m;
```

```
/* sum all the pair-dependencies */
CREATE TABLE bc
 AS SELECT F2.m AS m, sum(F2.count/F1.count) AS bc
 FROM F2
 JOIN F1 ON F2.i=F1.i AND F2.j=F1.j
 GROUP BY F2.m;
```

It is apparent from Equation 3.1 that all the shortest paths between each pair of vertexes are needed. So the diameter is needed to determine how many depths of JOIN are needed to go through in SQL queries to obtain all the shortest paths.

The join query is the same as doing a transitive closure except all the intermediate vertices need to be stored and each path needs to be given a unique id. Since the specified depth is equal or larger than the longest shortest path of the graph, the table U has all the shortest paths between each pair of vertices. Since each path has a unique id, the s_path in:

$$F1(i, j, s\_path) = \pi_{u,v,count(id)}(U),$$

should be the number of shortest paths for pair (i, j), and s_path_m in :

$$F2(i, j, m, s\_path\_m) = \pi_{i,j,m,count(id)}(U),$$

should be the number of shortest paths for pair (i, j) where m lies on the path. Then the s_path divided by s_path_m is the value of between centrality of vertex m.

4.2.2 Optimizing Queries

4.2.2.1 Diameter

Two different optimization algorithms for diameter have been developed. First, the focus reduced the number of doing JOIN. Suppose, the diameter of a graph is d, then at least a depth, d, is needed to get the result in the original method. Using logarithmic recursive queries, the diameter in $\log_2 d + 1$ depths can be obtained. $R, R^2, R^4 ... R^k$ are computed using similar JOIN condition $R.i = R.j$ as mentioned above, and k is equal to or greater than d. Another table $D, D^2, D^4 ... D^k$, which is obtained by $R^n \cup E$, is computed as

$$D^1 = R, D^2 = \sigma_{i,j,min(v)}(D^1 \cup R^2).$$

After each iteration, check the maximum length of all the shortest paths. The diameter is reached if this value does not change as the depth increases. The SQL query to do join and get $D(i, j, v)$ is given below.

```
SELECT Rd.i, Rd.j, Rd.v+Rd.v
 FROM  Rd JOIN Rd
 ON Rd.j=Rd.i
 GROUP BY Rd.i, Rd.j
 WHERE Rd.i < > Rd.j;


CREATE TABLE D
 AS SELECT i,j,min(v) FROM
```

```
 (SELECT * FROM T UNION ALL
   SELECT * FROM Rd)temp
 GROUP BY Rd.i, Rd.j;
```

Another way of optimizing the diameter algorithm is reducing the table size after each join. Only the shortest path is needed to get the diameter. Unnecessary paths are kept even if GROUP BY is used in each join in the original method. For example, the shortest path between one pair of vertices exists in depth 2. But there is another path for this pair in depth 3. Obviously, the second path is not the shortest path, but it is still kept and explored. This increases both space and time complexity. To keep only the shortest paths, we maintain one table $S(i, j, v)$ which contains all the shortest paths found so far, and delete unnecessary paths from the temporary join table, R_k, by comparing it with S after join. At the same time, we update table S by the addition of new shortest paths. Since DELETE query is slower than the CREATE query, we created a new table instead of deleting unnecessary paths. The JOIN query is the same as the original solution. The SQL query to delete unnecessary paths is given below. From this $R_n(i, j, v)$ table, table $S(i, j, v)$ is updated with an INSERT INTO SELECT statement.

```
/* JOIN */
CREATE TABLE Rt
 AS SELECT t1.i AS i, t2.j AS j, min(t1.v + t2.v) AS v
 FROM Rn t1
 JOIN R1 t2 on t1.j=t2.i
 GROUP BY t1.i, t2.j
```

```
 HAVING t1.i!=t2.j;


/*deleting unnecessary paths*/
CREATE TABLE  Rn  AS
 SELECT i, j, min(v) AS v
 FROM Rt
 WHERE (i,j,v) NOT IN
             (SELECT Rt.i, Rt.j, Rt.v FROM  Rt , S
              WHERE Rt.i =S.i AND Rt.j = S.j AND Rt.v > S.v)
 GROUP BY i,j;



 /* updating the table S */
INSERT INTO S
 SELECT  i,j,min(v) AS v
 FROM Rn
 GROUP BY i,j;
```

Here, we make sure that there was no potential shortest path deleted when deleting unnecessary paths. Contradiction was used to prove this.

We assumed there was a potential shortest path deleted. As Figure 4.3 shows, there are two paths from the source vertex S to the vertex D, and the distance of the path $w1$ is larger than that of $w2$. The shortest path from the source vertex S

to vertex U includes the path $w1$.

If $w1$ is deleted, the shortest path for pair(S,U) will be deleted. If $w1 + w3$ is the shortest path for pair(S,U), then:

```
w1 + w3 <= w2 + w3
w1 <= w2 .
```

This is impossible since $w1 > w2$. Thus, no potential shortest path will be deleted when unnecessary paths are deleted after each join.

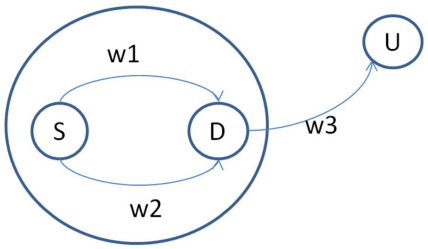

Figure 4.3: The figure for the contradiction proof.

4.2.2.2 Betweenness Centrality

The optimization of betweenness centrality is similar to the second way of optimizing diameter queries. A table S is continually updated in which the shortest paths found are stored, and unnecessary paths are deleted from the temporary join tables. The difference is that all the shortest paths and the intermediate vertices are kept. GROUP BY was not allowed in this part. The join query is given below. The other parts are the same as the original solution such as changing intermediate vertex

columns into rows, union all temporary tables together, using GROUP BY to get the pair-dependency, and summing them up.

```
CREATE TABLE Rn AS
 SELECT nextval('sequ') AS id, n AS d, t1.i AS i, t2.j AS j, t1.v+t2.v AS v,
         t1.m1 AS m1,t1.m2 AS m2,...,t1.m(n-2) AS m(n-2), t2.i AS m(n-1)
 FROM Rn-1 t1
 JOIN R1 t2 ON t1.j=t2.i;
```

4.3 Graph Partition

Vertica partitions for data is very efficient in parallel processing. It would be more feasible to maintain a duplicate copy of E (E_d) while performing self joins ($E \bowtie E$). A duplicate version of E can be maintained, and partition one E by i column and another E by j column when doing self joining on $E.j = E.i$. This will improve the join performance significantly. Partitioning divides one large table into smaller pieces based on values in one or more columns. Partitioning can improve parallelism during query execution and make other optimizations possible. The graph should be partitioned to reduce costly data movement across the network. This is possible when the parallel join occurs locally on each worker node. Partitioning provides opportunities for parallelism during query processing. Partition is different from segmentation. Segmentation helps split data evenly across nodes in a cluster, but partitioning helps organize data on each node into different storage containers. Therefor, partitioning would improve query performance.

Partitioning by vertex was performed. All the neighbors of a vertex were stored on the same machine. It was assumed that there were a few high degree vertices.

4.4 Time Complexity

The most challenging part to compute betweenness centrality and diameter is performing JOIN multiple times. The basic operation of iteratively perform JOIN is to multiply E by itself: $E \cdot E...E$. For the graphs stored in a database, $E \cdot E$ is matrix-matrix multiplication. The shape, density, and connectivity of a graph will impact the complexity of JOIN. Time and space complexity was analyzed for iterative matrix-matrix multiplication regarding different graph structures in the original and optimized solution.

First, we will focus on the $O()$ of $|R_2|$. For a tree graph, $|E \bowtie E| = O(n)$ since it was necessary to exclude the leaf nodes and the parents of leaf nodes. For a complete graph, $|E \bowtie E| = O(n^3)$ as there are $n(n-1)$ pairs of vertices, and there are $n-2$ paths for each pair. The ultimate goal was to understand $|R_k|$, where $R_k = E \bowtie E \bowtie ... \bowtie E$. To get the betweenness centrality value, it was necessary to go through k depth where $k >= p$, p is the longest shortest path for graphs (p equal to d for directed and connected graphs). The worst case is when the graph is a list, then $p = O(n)$. So p is the second aspect impacting $|R_k|$, and $|R_k|$ grows exponentially as k grows.

The optimization of betweenness centrality and the second optimization of diameter keep the shortest paths at each depth. Assume the number of shortest paths for

each pair is 1, then the total number of shortest paths for every pair is $O(n^2)$: then $|R_k| = O(n^2)$.

For the time complexity of the JOIN operator, it can range from $O(m)$ to $O(m^2)$ in each iteration. We checked the query plan in the DBMS, and it always used a hash join. The time complexity of a hash join could be as bad as $O(m^2)$ for a very dense graph but is $O(m)$ on average. For the second optimization method of diameter, since a logarithmic JOIN is performed, there are d times JOIN in the original solution. This is at most $\log_2(d) + 1$ times JOIN in the second optimization method.

Chapter 5

Experimental Evaluation

We have built SQL solutions for diameter and betweenness centrality based on the general algorithms. The solutions were optimized. In this section, experimental validations of these algorithms are proposed. First, an overview of the experimental setup and benchmark data sets are presented, followed by the accuracy validation. In each evaluating part, the correctness of the methods was shown by comparison to a Python package, followed by an evaluation of the impact of optimizations. Finally, we experimentally studied the performance of the algorithm compared with other solutions. Since Python and Spark are popular platforms for graph analytics, our queries were compared to Spark-GraphX on parallel machines and Python on a single machine.

5.1 DBMS Software and Hardware

All the systems were run on eight node clusters that each had an Intel Pentium(R) CPU running at 1.6 GHz, 8 GB of RAM, 1 TB disk, 224 kb L1 cache, 2 MB L2 cache and running Linux Ubuntu 14.04. For the experiments conducted in parallel computation, the total RAM size was 64 GB, and total disk memory was 8 TB. We used the Vertica DBMS supporting ANSI SQL to execute our queries. However, our queries were standard SPJ queries and work on other DBMSs.

5.2 Data Sets

Both synthetic and real-graph data sets were used for experimental evaluation. For synthetic-graph data sets, graphs were generated with varying complexity. Generated graphs with varying clique sizes used a uniform distribution where clique sizes increased linearly. The Stanford Network Analysis Platform (SNAP) repository was used for real data sets. The data sets are listed in Table 5.1. All the time measurements in this section are taken as the average of running each query five times and excluding the maximum and minimum values.

Table 5.1: Summary of data sets.

Dataset	Type	n	m	Skewed Vertices
tree10m	Synthetic	10 M	10 M	Low
cliqueLinear10m	Synthetic	48.5 k	10.2 M	High
cliqueLinear100k	Synthetic	2.3 k	100 k	High
wiki-vote	Real	8 k	103.6 k	Low
webgoogle	Real	875 k	5.1 M	Low

5.3 Accuracy Validation

In this section, we show our solution is computationally accurate since the results of our solution for different types of data sets both in Python and DBMS were identical.

5.3.1 Diameter

There is a function incorporated in Python-NetworkX ($diameter(G, e)$) that gives diameter values. Since the time needed to get diameters for real data sets was long because of large datasets, two types of subsets were selected from the graphs; graphs with larger indegree and graphs with smaller indegree. For dense subsets, a vertex with a large indegree were selected first. Then the graph was extended by choosing all the vertices connected to this vertex, repeating this process several times. For sparse subsets, one vertex whose indegree was only 1 was selected and then the graph was extended as was done for the dense graph. Finally for the other subsets, subWebgoogle and subWikivote, one vertex was randomly selected and then expanded as the sparse and dense subsets. These graphs were all connected graphs so that the diameters could be computed. The comparison results are presented in Table 5.2, along with the details of the data sets. The diameter given by our solution was the same as given by the Python function.

Table 5.2: Experimental proof for Diameter.

Data Set	m	n	Python	DBMS	Relative error(%)
subWebgoogle3	2.8 k	2.1 k	3	3	0
denseWebgoogle	37.3 k	7.4 k	4	4	0
sparseWebgoogle	22.3 k	5.6 k	11	11	0
subWikivote	34 k	3.6 k	5	5	0
clique100k	100.2 k	2.3 k	133	133	0

5.3.2 Betweenness Centrality

The definition of betweenness centrality could extend naturally to directed or disconnected graphs [5]. Along with the previously used subsets, another subset was added that is a disconnected subset named sparseWebgoogle2 (no diameter for this graph). This was created by randomly selecting some vertices whose indegree were only 1. A function ($betweenness\_centrality(G, k, normalized, weight, endpoints, seed)$) was incorporated in Python-NetworkX that gave the shortest-path betweenness centrality for vertices. Table 5.3 shows the comparison results. The maximum relative error among all the vertices in the test graph is shown in Table 5.3. The depth was set equal to the length of the longest shortest path to make sure of getting the correct betweenness centrality for each vertex. The relative errors of our algorithm are lower than 0.0003 percent for all the data sets from Table 5.3. The relative error results from rounding, a different adding order when adding a very large number with a small number.

Figure 5.1 - 5.3 indicate the detail of betweenness centrality results for the different data sets. If a vertex's betweenness centrality value is 0, then this vertex id

Table 5.3: Experimental proof for Betweenness Centrality.

Data Set	m	n	Depth	relative error
subWebgoogle3	2.8 k	2.1 k	3	0
subWebgoogle4	18.4 k	5.2 k	22	1E-9
denseWebgoogle	37.3 k	7.0 k	5	0–0.0003
sparseWebgoogle2	100 k	153.4 k	5	0
subWikivote	34 k	3.6 k	5	1E-05

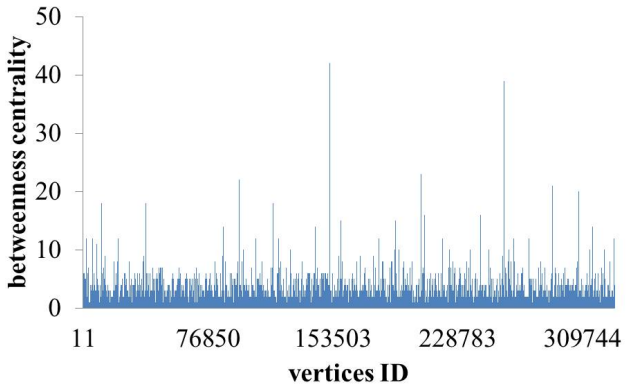

Figure 5.1: The Betweenness Centrality value for sparseWebgoogle2.

is not shown in the figures. We already know that subWikivote and subWebgoogle4 are tree-structure graphs. While leaf nodes of a tree have 0 betweenness centrality, the root has the maxium value of betweenness centrality. As the depth of the tree increases, the number of nodes significantly increases, while the betweenness centrality for these nodes significantly decreases. This is exactly the situation demonstrated in Figure 5.3 and 5.2. For the sparseWebgoogle2 dataset, the difference of betweenness centrality among all the vertices are very small and most of the vertices have low betweenness centrality values. Figure 5.1 demonstrates sparseWebgoogle2 is a sparse graph.

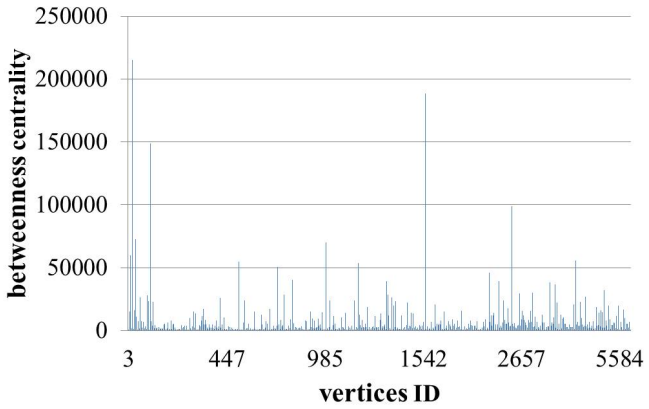

Figure 5.2: The Betweenness Centrality value for subWebgoogle4.

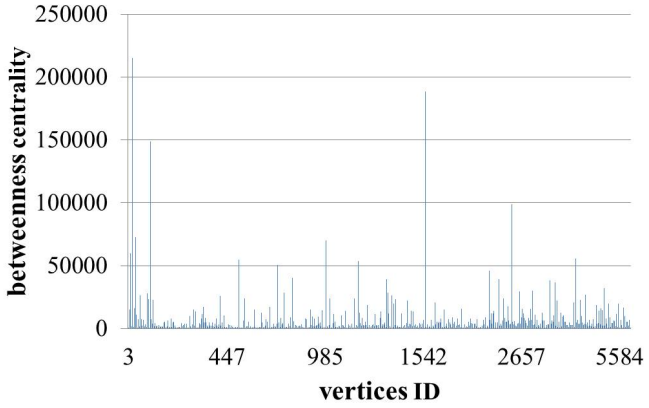

Figure 5.3: The Betweenness Centrality value for subWiki.

5.4 Evaluation of the Impact of Optimizations

5.4.1 Diameter

In this solution, the optimized algorithms are compared with the original algorithm for diameter. Table 5.4 shows the comparison results. A "Stop" sign is placed in the table if the time is more than 30 min. While $\log D$ in the table means the optimization doing the logarithmic join, sJOIN deletes unnecessary paths. The sJOIN is the fastest, while $\log D$ is faster than the original solution for data sets with large diameters but slower for data sets with small diameters as seen in Table 5.4. This is because the table size in $\log D$ is larger than the original solution. Based on the experiments, sJOIN is the default optimization for the diameter algorithm.

Table 5.4: Evaluating the impact of optimization for Diameter (time in seconds).

Data Set	$\log_D$	sJOIN	original
subWebgoogle3	319 s	11 s	15 s
denseWebgoogle	Stop	165 s	170 s
sparseWebgoogle	1309 s	60 s	234 s
clique100k	510 s	246 s	1016 s
subWikivote	300 s	47 s	55 s

5.4.2 Betweenness Centrality

Experiments were performed both with and without optimization for the betweenness centrality algorithm. Table 5.4 shows that the solution with optimization takes almost the same time with the method without optimization for sparse and small

graphs, but much less time is for dense and large graphs, because, the number of unnecessary paths is small for sparse and small graphs and large for dense graphs. Deleting those unnecessary paths, the performance for dense graphs was improved. Based on the experiments, this is the default optimization for betweenness centrality algorithms. Figure 5.4 shows the table size at different depths for the original solution and the optimized solution. In Figure 5.4, the table size grows dramatically with increasing depth in the original solution. The table size keeps the same size or grows only slightly in the optimized solution. $|R_k|$ in the original and optimized solution could be expressed as $x^k n$ when k is within some range, where k is the depth of join, and n is the number of vertices in the E. The values of x for different graphs are listed in Table 5.6. $|R_k|$ grows much slower in the optimized algorithm when compared with the original solution, Table 5.6.

Table 5.5: Evaluating the impact of optimization for Betweenness Centrality (time in seconds).

table name	without opt	with opt	Depth
subWebgoogle3	4 s	5 s	3
subWebgoogle4	Stop	1562 s	22
denseWebgoogle	Stop	1558 s	5
sparseWebgoogle2	6 s	9 s	5
subWikivote	Stop	345 s	5

Table 5.6: The average value of x for different data sets.

Data set	n	m	x in original	x in optimization
subWebgoogle4	5.2 k	18.4 k	11	2.8
webGoogle	875 k	5.1 M	12	6
wikivote	103.6 k	8 k	43	16
denseWebgoogle	7.4 k	37.3 k	15.5	6

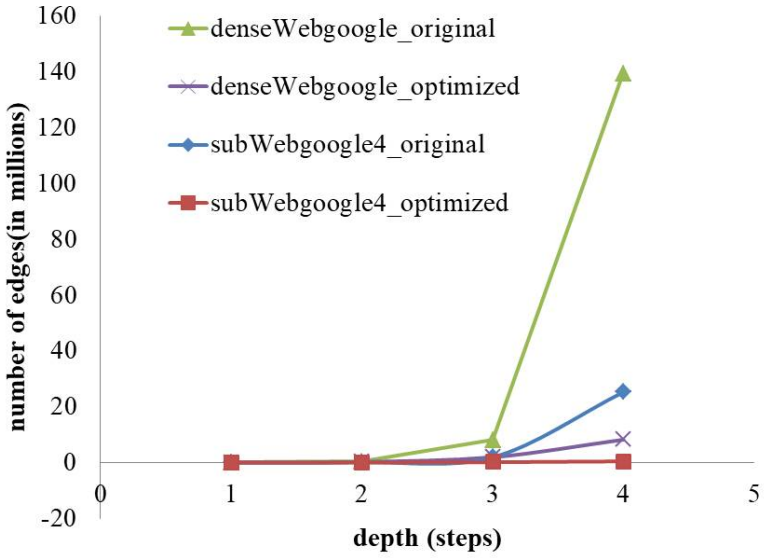

Figure 5.4: How table sizes change when doing join in the original solution and optimized solution.

Table 5.7: Time to compute Diameter in single machines (time in seconds).

Data set	DBMS	Python	Diameter
subWebgoogle3	21 s	69 s	3
denseWebgoogle	396 s	1472 s	4
sparseWebgoogle	207 s	751 s	11
subwikivote	138 s	434 s	5
clique100k	600 s	305 s	133

5.5 Performance Comparison

5.5.1 Diameter

The algorithm was compared to the Python solution running on a single machine of the server, and compared to Spark running on an eight node cluster. The same data sets were used for the accuracy validation. Table 5.7 shows the single machine results. For our solution was faster than Python for all data sets except clique100k, because the number of joins required for this graph is high.

For parallel processing, there is a function incorporated in Spark-GraphX (*shortest-path*()) which calculates the shortest paths of all pairs of vertices. After which, the longest shortest path is chosen as the diameter. DBMS is faster than Spark for sparse graphs Table 5.8, while for dense or large graphs, DBMS could give the results in a short time while Spark crashed during computation because it ran out of memory. Spark has a main memory limitation, but our solution removes the RAM limitation.

Table 5.8: Time to compute Diameter in parallel machines: DBMS vs Spark (time in seconds).

Data Set	Spark	DBMS	D/S
subWebgoogle3	43 s	11 s	0.26
denseWebgoogle	Crashed	165 s	–
sparseWebgoogle	135 s	60 s	0.44
subwikivote	Crashed	47 s	–
clique100k	Crashed	246 s	–

5.5.2 Betweenness Centrality

Besides the full betweenness centrality, $k-$betweenness centrality captures information provided by paths whose length is within k unions of the shortest path length, and is also a useful kernel for analyzing the importance of vertices. While Python can only give an exact and approximate value of full betweenness, our solution can provide both the full and $k-$betweenness centrality. To compare our solution against Python, the full betweenness centrality was used with the same graphs as the accuracy validation part, while the depth was the same. Table 5.9 demonstrates the comparison results. A "Stop" was placed if one computation did not finished in 30 min. Table 5.9 shows that Python was faster than DBMS for dense graphs, especially for those graphs having longer diameter. This is because DBMS needed to go through many depths. However, for sparse graphs, DBMS was much faster than Python due to the small table size when performing joins.

The available betweenness centrality solution for parallel processing in Spark was compared to our method. Both solutions were run on the eight node cluster. It

Table 5.9: Time to compute Betweenness Centrality in single machine: DBMS vs Python (time in seconds).

Data Set	Python	DBMS	D/P
subWebgoogle3	6 s	1 s	0.18
subWebgoogle4	153 s	4517 s	29.5
denseWebgoogle	621 s	3846 s	6.2
sparseWebgoogle2	Stop	3 s	–
subWikivote	76 s	752 s	9.89

Table 5.10: Time to compute Betweenness Centrality in parallel machines: DBMS vs Spark (time in seconds).

	depth=4		depth=5		depth=6	
Data set	DBMS	Spark	DBMS	Spark	DBMS	Spark
subWebgoogle4	205 s	Crashed	912 d	Crashed	1480 s	Crashed
denseWebgoogle	1558 s	Crashed	1680 s	Crashed	1775 s	Crashed
sparseWebgoogle2	9 s	34 s	10 s	39 s	12 s	44 s
clique100k	25 s	Crashed	29 s	Crashed	37 s	Crashed

was possible to compute $k-$betweenness centrality in Spark with a calcuation up to a certain depth where depth k was 4,5,6. Table 5.10 shows the calculation time. "Crashed" was inserted when Spark crashed because it ran out of memory. From Table 5.10, the Spark was slower than DBMS for sparse graphs and crashed for dense and large graphs because Spark has the main memory limitation. DBMS computed $k-$betweenness centrality for both dense and sparse graphs in a reasonable time.

To summarize this section, the proposed algorithms implemented with SQL queries in DBMS were compared with popular existing systems, demonstrating that our algorithms are accurate and faster.

Chapter 6

Conclusion

6.1 Conclusion

1. This study demonstrates that parallel DBMS can indeed help to compute something more complicated than transitive closure and recursive queries. The solution has no main memory limitation for parallel columnar DBMS.

2. This study expresses the computation of diameter and betweenness centrality with queries and proposes several optimization methods on the queries.

3. Optimizing the query performance helped to compute the algorithms faster than usual.

4. The experimental results indicate that the study's queries performed better in most cases than Python on one machine and Spark-GraphX in parallel machines. We also provide accuracy proof that this study's solution was computationally accurate.

However, one limitation of the proposed solution is too many join operations for betweenness centrality slow down the computation in very dense graphs, and that the solution was tested on finite graphs.

6.2 Future Work

Based on the results of this study, the future work should include; should sample to get the approximate value of diameter and betweenness centrality, compute graph circumference (longest cycle), detecte two or more disconnected subgraphs, check if the graph contains cliques of size at least $k \geq 3$, count maximal cliques, parallel speedup or how the queries perform when the number of machines is varied, and discover complex patterns beyond paths. Moreover, this study plans to optimize the algorithms in Spark.

Bibliography

[1] T. Abughofa and F. Zulkernine. Sprouter: Dynamic graph processing over data streams at scale. In *International Conference on Database and Expert Systems Applications(2)*, pages 321–328, 2018.

[2] S. T. Al-Amin, C. Ordonez, and L. Bellatreche. Big data analytics: Exploring graphs with optimized sql queries. In *Database and Expert Systems Applications*, pages 88–100, 2018.

[3] M. Alemi, H. Haghighi, and S. Shahrivari. Ccfinder: using spark to find clustering coefficient in big graphs. *The Journal of Supercomputing*, 73(11):4683–4710, 2017.

[4] J. Anthonisse. The rush in a directed graph. *Stichting Mathematisch Centrum. Mathematische Besliskunde, No. BN 9/71.*, pages 163–177, 1971.

[5] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. In *Algorithms and Models for the Web-Graph*, pages 124–137, 2007.

[6] J. Balaji and R. Sunderraman. Distributed graph path queries using spark. In *40th IEEE Annual Computer Software and Applications Conference*, pages 326–331, 2016.

[7] M. Barthélemy. Betweenness centrality in large complex networks. *The European Physical Journal B*, 38(2):163–168, 2004.

[8] M. Bertolucci, A. Lulli, and L. Ricci. Current flow betweenness centrality with apache spark. In *Algorithms and Architectures for Parallel Processing*, pages 270–278, 2016.

[9] M. Biskup. Graph diameter in long-range percolation. *Random Struct. Algorithms*, 39(2):210–227, 2011.

[10] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25:163–177, 2001.

[11] W. Cabrera and C. Ordonez. Scalable parallel graph algorithms with matrix-vector multiplication evaluated with queries. *Distributed and Parallel Databases*, 35(3-4):335–362, 2017.

[12] S. Chechik, D. H. Larkin, L. Roditty, G. Schoenebeck, R. E. Tarjan, and V. V. Williams. Better approximation algorithms for the graph diameter. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1041–1052, 2014.

[13] Y. Chen. On the evaluation of large and sparse graph reachability queries. In *International Conference on Database and Expert Systems Applications*, pages 97–105, 2008.

[14] Y. Cho. A parallel community detection in multi-modal social network with apache spark. *IEEE Access*, 7:27465–27478, 2019.

[15] L. C. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40(1):35–41, 1977.

[16] R. Geisberger, P. Sanders, and D. Schultes. Better approximation of betweenness centrality. In *Proceedings of the Meeting on Algorithm Engineering and Expermi-ments*, pages 90–100, 2008.

[17] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. Vertexica: Your relational friend for graph analytics! *Proceedings of the Very Large Database Endowment.*, 7(13):1669–1672, Aug. 2014.

[18] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146. ACM, 2010.

[19] Y. A. Megid, N. El-Tazi, and A. Fahmy. Using functional dependencies in conversion of relational databases to graph databases. In *International Conference on Database and Expert Systems Applications(2)*, pages 350–357, 2018.

[20] H. Naacke, B. Amann, and O. Curé. SPARQL international graph pattern processing with apache spark. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences and Systems, GRADES@SIGMOD/ PODS 2017*, pages 1:1–1:7, 2017.

[21] C. Ordonez. Optimization of linear recursive queries in SQL. *IEEE Trans. Knowl. Data Eng.*, 22(2):264–277, 2010.

[22] C. Ordonez, W. Cabrera, and A. Gurram. Comparing columnar, row and array dbmss to process recursive queries on graphs. *Information Systems*, 63:66–79, 2017.

[23] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 165–178, 2009.

[24] M. A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, pages 1–10, 2015.

[25] H. Thakkar, D. Punjani, S. Auer, and M.-E. Vidal. Towards an integrated graph algebra for graph pattern matching with gremlin. In *Database and Expert Systems Applications*, pages 81–91, 2017.

[26] K. Zhao and J. X. Yu. All-in-one: Graph processing in rdbmss revisited. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1165–1180, 2017.